
The Symbolic Interior Point Method

Martin Mladenov
TU Dortmund University
martin.mladenov@cs.tu-dortmund.de

Vaishak Belle
KU Leuven
vaishak@cs.kuleuven.be

Kristian Kersting
TU Dortmund University
kristian.kersting@cs.tu-dortmund.de

Abstract

A recent trend in probabilistic inference emphasizes the codification of models in a formal syntax, with suitable high-level features such as individuals, relations, and connectives, enabling descriptive clarity, succinctness and circumventing the need for the modeler to engineer a custom solver. Unfortunately, bringing these linguistic and pragmatic benefits to numerical optimization has proven surprisingly challenging. In this paper, we turn to these challenges: we introduce a rich modeling language, for which an interior-point method computes approximate solutions in a generic way. While logical features easily complicates the underlying model, often yielding intricate dependencies, we exploit and cache local structure using algebraic decision diagrams (ADDs). Indeed, standard matrix-vector algebra is efficiently realizable in ADDs, but we argue and show that well-known second-order methods are not ideal for ADDs. Our engine, therefore, invokes a sophisticated matrix-free approach. We demonstrate the flexibility of the resulting symbolic-numeric optimizer on decision making and compressed sensing tasks with millions of non-zero entries.

1 Introduction

Numerical optimization is arguably the most prominent computational framework in machine learning and AI. It can be seen as an *assembly language* for hard combinatorial problems ranging from classification and regression in learning, to computing optimal policies and equilibria in decision theory, to entropy minimization in information sciences [1, 2, 3]. What makes optimization particularly ubiquitous is that, similar to probabilistic inference, it provides a formal apparatus to reason (*i.e.* express preferences, costs, utilities) about possible worlds. However, it is also widely acknowledged that many AI models are often best described using a combination of natural and mathematical language and, in turn, require algorithms to be individually engineered.

An emerging discipline in probabilistic inference emphasizes the codification of models in a suitable formal syntax [4, 5, 6], enabling the rapid prototyping and solving of complex probabilistic structures. They are driven by the following desiderata:

- (D1) The syntax should be expressive, allowing high-level (logical) features such as individuals, relations, functions, connectives, in service of descriptive clarity and succinct characterizations, including context-specific dependencies (*e.g.* pixels probabilistically depend only on neighboring pixels, the flow of quantities at a node is limited to connected nodes).
- (D2) The inference engine should be *generic*, circumventing the need for the modeler to develop a custom solver.

Unfortunately, bringing these linguistic and pragmatic benefits to optimization has proven surprisingly challenging. Mathematical programs are specified using linear arithmetic, matrix and tensor algebra, multivariate functions and their compositions. Classes of programs (*e.g.* linear, geometric) have

distinct standard forms, and solvers require models to be painstakingly reduced to the standard form; otherwise, a custom solver has to be engineered. The approach taken in recent influential proposals, such as disciplined programming [7], is to carefully constrain the specification language so as to provide a structured interface between the model and the solver, by means of which geometric properties such as the curvature of the objective can be inferred. Basically, when contrasted with the above desiderata, solver genericity is addressed only partially, as it requires the modeler to be well-versed in matrix-vector algebra, and little can be said about high-level features, *e.g.* for codifying dependencies.

In this paper, we are addressing the above desiderata. Doing so has the potential to greatly simplify the specification and prototyping of AI and ML models [8, 1].

Let us elaborate on the desiderata in our context. The introduction of logical features, while attractive from a modeling viewpoint, assuredly complicates the underlying model, yielding, for example, intricate dependencies. Local structure, on the other hand, can be exploited [9]. In probabilistic inference, local structure has enabled tractability in high-treewidth models, culminating in the promising direction of using circuits and decision-diagrams for the underlying graphical model [10]. Such data structures are a more powerful representation, in admitting compositionality and refinement. Here, we would like to consider the idea of using such a data structure for optimization, for which we turn to early pioneering research on *algebraic decision diagrams* (ADDs) that supports efficient matrix manipulations (compositionality) and caching of submatrices (repeated local structure) [11, 12].

Now, suppose a linear program encoded in a high-level language has been reduced to an efficiently manipulable data structure such as an ADD. Unfortunately, it is far from obvious how a generic solver can be engineered for it. Matrix operations with ADDs, for example, are efficient only under certain conditions, such as: a) they have to be done recursively, in a specific descent order; b) they have to involve the entire matrix (batch mode), *i.e.* access to arbitrary submatrices is not efficient. This places rather specific constraints on the kind of method that could benefit from an ADD representation, ruling out approaches like random coordinate descent. In this paper, we aim to engineer and construct a solver for such matrices in circuit representation. We employ ideas from the matrix-free interior point method [13], which appeals to an iterative linear equation solver together with the log-barrier method, to achieve a regime where the constraint matrix is only accessed through matrix-vector multiplications. Specifically, we show a ADD-based realization of the approach leverages the desirable properties of these representations (*e.g.* caching of submatrices), leading to a robust and fast engine. We demonstrate this claim empirically.

2 Lineage

Expressiveness in modeling languages for numerical optimization is a focus of many proposals, *e.g.* [14, 15]. However, they blur the border between declarative and imperative paradigms, using sets of objects to index LP variables and do not embody logical reasoning. Disciplined programming [7] enables an object-oriented approach to constructing optimization problems, but falls short of our desiderata as argued before. Taking our cue from statistical relational learning [16, 17], we argue for algebraic modeling that is fully integrated with classical logical machinery (and not just logic programming [18]). This allows the specification of correlations, groups and properties in a natural manner, as also observed elsewhere [19, 20].

The efficiency of ADDs for matrix-vector algebra was established in [12]. In particular, the use of ADDs for compactly specifying (and solving) Markov decision processes (*i.e.* representing transitions and rewards as Boolean functions) was popularized in [21]; see [22, 23] for recent offerings. We differ fundamentally from these strands of work in that we are advocating the realization of iterative methods using ADDs, which (surprisingly) has never been studied in great detail to the best of our knowledge. Therefore, we call our approach *symbolic numerical optimization*.

3 Primer on Logic and Decision Diagrams

We cover some of the logical preliminaries in this section. To prepare for the syntax of our high-level mathematical programming language, we recap basic notions from mathematical logic [24]. A propositional language \mathcal{L} consists of finitely many propositions $\mathcal{P} = \{p, \dots, q\}$, from which formulas are built using connectives $\{\neg, \vee, \wedge\}$. A \mathcal{L} -model M is a $\{0, 1\}$ -assignment to the symbols in \mathcal{P} ,

which is extended to complex formulas inductively. For example, if $\mathcal{P} = \{p, q\}$ and $M = \langle p = 1, q = 0 \rangle$, we have $M \models p$, $M \models p \vee q$, $M \models \bar{q}$ but $M \not\models p \wedge q$.

The logical language of finite-domain function-free first-order logic consists of finitely many predicate symbols $\{P(x), \dots, Q(x, y), \dots, R(x, y, z), \dots\}$ and a domain D of constants. Atoms are obtained by substituting variables in predicates with constants from D , e.g. $P(a), Q(a, b), R(a, b, c)$ and so on wrt $D = \{a, b, \dots, c\}$. Formulas are built as usual using connectives $\{\neg, \vee, \wedge, \forall, \exists\}$. Models are $\{0, 1\}$ -assignments to atoms, but can also interpret quantified formulas, e.g. $\forall x[P(x)]$. For example, if $D = \{0, 1\}$ and $M = \langle P(0) = 1, P(1) = 1 \rangle$, then $M \models P(0)$, $M \models P(0) \wedge P(1)$ and $M \models \forall x P(x)$. Finally, although finite-domain function-free first-order logic is essentially propositional, it can nonetheless serve as a convenient template for specifying correlations, groups and properties, cf. statistical relational learning [16].

A BDD [25] is a compact and efficiently manipulable data structure for a Boolean function $f: \{0, 1\}^n \rightarrow \{0, 1\}$. Its roots are obtained by the Shannon expansion of the *cofactors* of the function: if f_x and $f_{\bar{x}}$ denote the partial evaluation of $f(x, \dots)$ by setting the variable x to 1 and 0 respectively, then

$$f = x \cdot f_x + \bar{x} \cdot f_{\bar{x}}.$$

When the Shannon expansion is carried out recursively, we obtain a full binary tree whose non-terminal nodes, labeled by variables $\{\dots, x_i, \dots\}$, represent a function: its left child is f 's cofactor w.r.t. x_i for some i and its right child is f 's cofactor w.r.t. \bar{x}_i . The terminal node, then, is labeled 0 or 1 and corresponds to a total evaluation of f . By further ordering the variables, a graph, which we call the (ordered) BDD of f , can be constructed such that at the k th level of the tree, the cofactors wrt the k th variable are taken. Given an ordering, BDD representations are *canonical*: for any two functions $f, g: \{0, 1\}^n \rightarrow \{0, 1\}$, $f \equiv g \Leftrightarrow f_x \equiv g_x$ and $f_{\bar{x}} \equiv g_{\bar{x}}$; and *compact* for Binary operators $\circ \in \{+, \times, \dots\}$: $|f \circ g| \leq |f| |g|$.

ADDs generalize BDDs in representing functions of the form $\{0, 1\}^n \rightarrow \mathbb{R}$, and so inherit the same structural properties as BDDs except, of course, that terminal nodes are labeled with real numbers [11, 12]. Consider any real-valued vector of length m : the vector is indexed by $\lg m$ bits, and so a function of the form $\{0, 1\}^{\lg m} \rightarrow \mathbb{R}$ maps the vector's indices to its range. Thus, an ADD can represent the vector. By extension, any real-valued $2^m \times 2^n$ matrix A with row index bits $\{x_1, \dots, x_m\}$ and column index bits $\{y_1, \dots, y_n\}$ can be represented as a function $f(x_1, y_1, x_2, \dots)$ such that its cofactors are the entries of the matrix. The intuition is to treat x_1 as the most significant bit, and x_m as the least. Then A represented by a function f as an ADD is: $(f_{\bar{x}_1 y_1} \ f_{\bar{x}_1 y_2})$ as the first row and $(f_{x_1 y_1} \ f_{x_1 y_2})$ as the second, where each submatrix is similarly defined wrt the next significant bit. Analogously, when multiplying two m -length vectors represented as Boolean functions $f, g: \{0, 1\}^{\lg m} \rightarrow \mathbb{R}$, we can write $[f_{\bar{x}_1} \ f_{x_1}][g_{\bar{x}_1} \ g_{x_1}]^T$, taking, as usual, x_1 as the most significant bit.

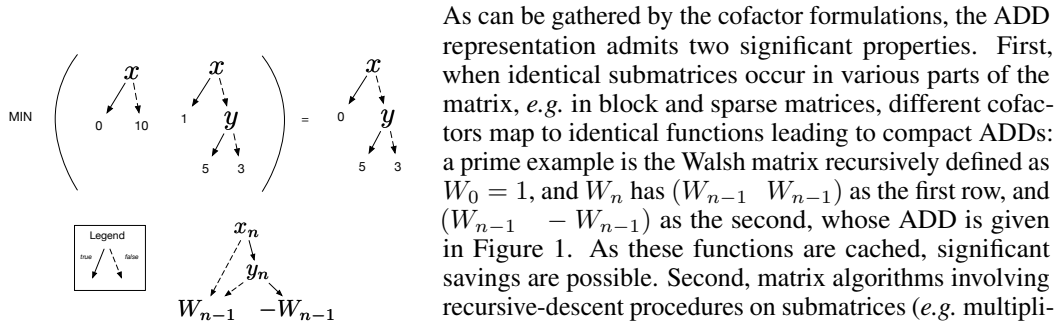


Figure 1: Minimization of two functions using ADDs (above), and the ADD for the Walsh matrix (below).

As can be gathered by the cofactor formulations, the ADD representation admits two significant properties. First, when identical submatrices occur in various parts of the matrix, e.g. in block and sparse matrices, different cofactors map to identical functions leading to compact ADDs: a prime example is the Walsh matrix recursively defined as $W_0 = 1$, and W_n has $(W_{n-1} \ W_{n-1})$ as the first row, and $(W_{n-1} \ -W_{n-1})$ as the second, whose ADD is given in Figure 1. As these functions are cached, significant savings are possible. Second, matrix algorithms involving recursive-descent procedures on submatrices (e.g. multiplication) and term operations (addition, maximization) are efficiently implementable by manipulating the ADD representation; see Figure 1. In sum, standard matrix-vector algebra can be implemented over ADDs efficiently, and in particular, the caching of submatrices in recursively defined operations (while implicitly respecting the variable ordering of the ADD) will best exploit the superiority of the ADD representation.

4 First-Order Logical Quadratic Programming

A convex quadratic program (QP) is an optimization problem over the space \mathbb{R}^n , that is, we want to find a real-valued vector $x \in \mathbb{R}^n$ from the solution set of a system of linear inequalities $\{x \geq 0 : Ax = b\}$ such that a convex quadratic objective function $f(x) = x^T Qx + c^T x$ is minimized. For each QP, there exists a complementary (dual) QP-D such that each solution of QP-D provides an upper bound on the minimum of QP, for the maximizer of QP-D, this bound is tight. In this paper we assume that QP and QP-D can be reduced to the following standard form

<p>PRIMAL-QP :</p> <p>minimize $c^T x + 1/2x^T Qx$</p> <p>subject to $Ax = b,$</p> <p style="padding-left: 2em;">$x \geq 0.$</p>	<p>DUAL-QP :</p> <p>maximize $b^T y - 1/2x^T Qx$</p> <p>subject to $A^T y + s - Qx = c,$</p> <p style="padding-left: 2em;">$s \geq 0.$</p>
---	---

Here, Q is positive semi-definite. Whenever $Q = 0$, we speak of linear programs (LPs).

In this paper, we would like to provide an expressive modeling language with high-level (logical) features such as individuals, relations, functions, and connectives. While (some) high-level features are prominent in many optimization packages such as AMPL [26], they are reduced (naively) to canonical forms transparently to the user, and do not embody logical reasoning. Logic programming is supported in other proposals [18], but their restricted use of negation makes it difficult to understand the implications when modeling a domain. We take our cue from statistical relational learning [16], as considered in [19], to support any (finite) fragment of first-order logic with classical interpretations for operators. Given a logical language \mathcal{L} , the syntax for first-order logical quadratic programs is:

$$\min_v. \quad \sum_{\{x, x' : \varphi(x, x')\}} q(x, x')v(x)v(x') + \sum_{\{x : \psi(x)\}} c(x)v(x) \quad \text{s.t.} \quad \{y : \psi(y)\} : \sum_{\{x : \phi(x, y)\}} a(x, y)v(x) \geq b(y)$$

where, we write $\delta(z)$ to mean that the \mathcal{L} -formula δ mentions the logical symbols z , and read $\{z : \delta(z)\}$ as the set of all assignments to z satisfying $\delta(z)$. The constraints are to be read procedurally as follows: for every assignment to y satisfying $\psi(y)$, consider the constraint $\sum_C a(x, y)v(x) \geq b(y)$, where the set C are those assignments to x satisfying $\phi(x, y)$. For example, consider:

$$\text{minimize}_v \quad \sum_{\{x : \text{TRUE}\}} v(x) \quad \text{subject to} \quad \{y : \text{TRUE}\} : \sum_{\{x : x \vee y\}} v(x) \geq 1, \quad \{y : \text{TRUE}\} : v(y) \geq 0.$$

That is, we are to minimize $v(x) + v(\bar{x})$ subject to $v(x) \geq 1$ and $v(\bar{x}) \geq 1$ (for the cases where \bar{y}), $v(x) + v(\bar{x}) \geq 1$ and $v(x) \geq 0$ (for the cases where y), yielding the following canonical form:

$$A = \begin{bmatrix} 0 & (\bar{x}\bar{y}) & 1 & (x\bar{y}) \\ 1 & (\bar{x}y) & 1 & (xy) \\ 1 & (\bar{x}y) & 0 & (xy) \\ 0 & (\bar{x}y) & 1 & (xy) \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad c = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

By extension, an example for a relational constraint is one of the form: $\sum_{x:\exists z \text{ Friends}(z, x)} v(x)$. Here, wrt $D = \{a, b, c\}$ and a database $\{\text{Friends}(b, a), \text{Friends}(b, c)\}$, we would instantiate the constraint as $v(b) + v(c)$. For simplicity, however, we limit discussions to propositions in the sequel.

One of the key contributions in our engine is that we are able to transform the constraints in such a high-level mathematical program directly to an ADD representation. (Quantifiers are eliminated as usual: existentials as disjunctions and universals as conjunctions [24].) The rough idea is this. Order the variables y in $\psi(y)$. Any propositional formula can be seen as a Boolean function mapping to $\{0, 1\}$. That is, assignments to y determine the value of the corresponding Boolean function, e.g. $\langle y_1 = 1, y_2 = 0 \rangle$ maps the function $y_1 \wedge y_2$ to 0. We build the ADD for $\psi(y)$, and for each complete evaluation of this function, we build the ADD for $\phi(x, y)$ and let its leaf nodes be mapped to $b(y)$. We omit a full description of this procedure, but go over the main result:

Theorem 1: *There is an algorithm for building an ADD for terms from a first-order logical mathematical program without converting the program to the canonical (ground) form. The ADD obtained from the algorithm is identical to the one obtained from the ground form.*

Proof: We simply need to argue that a term of a first-order logical mathematical program corresponds to a Boolean function $\{0, 1\}^n \rightarrow \mathbb{R}$, where n is the number of propositions. The procedure described above builds its ADD. By the canonicity of ADDs, the claim follows. ■

5 Solution Strategies for First-Order Logical QPs

Given the representation language, we now turn towards solving logical QPs. To prepare for discussions, let us establish an elementary notion of algorithmic correctness for ADD implementations. We assume that given a first-order logical mathematical program, we have in hand the ADDs for A , b and c . Then, it can be shown:

Theorem 2: *Suppose A , b and c are as above, and e is any arithmetic expression over them involving standard matrix binary operators. Then there is a sequence of operations over their ADDs, yielding a function h , such that $h = e$.*

The kinds of expressions we have in mind are $e = Ax - b$ (which corresponds to the residual in the corresponding system of linear equations). The proof is as follows:

Proof: For any $f, g : \{0, 1\}^n \rightarrow \mathbb{R}$, and (standard) binary matrix operators (multiplication, summation, subtraction), observe that $h = f \circ g$ iff $h_x = f_x \circ g_x$ and $h_{\bar{x}} = f_{\bar{x}} \circ g_{\bar{x}}$. By canonicity, the ADD for h is precisely the same as the one for f and g composed over \circ . In other words, for any arithmetic expression e over $\{f, g, \circ\}$, the ADD realization for h is precisely the same function. ■

To guide the construction of the engine, we will briefly go over the operations previously established as efficient with ADDs [11, 12], and some implications thereof for a solver strategy.

Theorem 3: [11, 12] *Suppose A, A' are real-valued matrices. Then the following can be efficiently implemented in ADDs using recursive-decent procedures: 1) accessing and setting a submatrix A^* of A ; 2) termwise operations, i.e. $(A \circ A')_{ij} = A_{ij} \circ A'_{ij}$ for any termwise operator \circ ; and 3) vector and matrix multiplications.*

The proof for these claims and the ones in the corollary below always proceed by leveraging the Shannon expansion for the corresponding Boolean functions, as shown in Section 3. We refer interested readers to [11, 12] for the complexity-theoretic properties of these operations. It is worth noting, for example, that multiplication procedures that perform both (naive) block computations and ones based on Strassen products can be recursively defined. For our purposes, we get:

Corollary 4: *Suppose $d = [d_1 \ \cdots \ d_m]$ and e are m -length real-valued vectors, and k is a scalar quantity. Then the following can be efficiently implemented in ADDs using recursive-descent procedures: 1) scalar multiplication, e.g. $k \cdot d$; 2) vector arithmetic, e.g. $d + e$; 3) element sum, e.g. $\sum_i d_i$; 4) element function application, e.g. if $w : \mathbb{R} \rightarrow \mathbb{R}$, then computing $w(d) = [w(d_1) \ \cdots \ w(d_m)]$; and 5) norms, e.g. $\|d\|$ and $\|d\|^2$.*

5.1 A Naive Ground-and-Solve Method

The most straightforward approach for solving a first-order logical QP (FOQP) is to reduce the problem to the normal form and use a standard solver for QPs, such as an interior point method, an augmented Lagrangian method, or some form of active set method, e.g. generalized simplex. The correctness of this solution strategy is guaranteed by the semantics of the first-order constraints: in general, every FOQP can be brought to the normal form. While this method would work, it exhibits a significant drawback in that the optimization engine cannot leverage knowledge about the symbolic structure of the problem. That is, even if the problem compiles to a very small ADD, the running time of the optimizer will depend (linearly at best) on the number of nonzeros in the ground matrix. Clearly, large dense problems will be completely intractable. However, as we will see later, some dense problems can still be attacked with the help of structure.

Input: (x^0, y^0, s^0) with $(x^0, s^0) \geq 0$
 $k \leftarrow 0$;
while *stopping criterion not fulfilled* **do**
 Solve (2) with $(x, y, s) = (x^k, y^k, s^k)$ to obtain a direction
 $(\Delta x^k, \Delta y^k, \Delta s^k)$;
 Choose step length $\alpha_k \in (0, 1]$;
 Update $(x^{k+1}, y^{k+1}, s^{k+1}) \leftarrow \alpha_k (\Delta x^k, \Delta y^k, \Delta s^k)$
 $k \leftarrow k + 1$
return x^k ;

(a) Primal-Dual Barrier Method

Input: $A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$
 $k \leftarrow 0, r_0 \leftarrow b - Ax_0$;
while *stopping criterion not fulfilled* **do**
 $k \leftarrow k + 1$;
 if $k = 1$ **then**
 $p_0 \leftarrow r_0$;
 else
 $\tau_{k-1} = (r_{k-1}^T r_{k-1}) / (r_{k-2}^T r_{k-2})$;
 $p_k = r_{k-1} + \tau_{k-1} p_{k-1}$;
 $\mu_k = (r_{k-1}^T r_{k-1}) / (p_k^T A p_k)$;
 $x^k = x_{k-1} + \mu_k p_k$;
 $r^k = r_{k-1} - \mu_k A p_k$;
 return x^k ;

(b) Conjugate Gradient Method

5.2 The Symbolic Interior Point Method

While the ground-and-solve method is indeed correct, one can do significantly better, as we will now show. In this section, we will construct a solver that automatically exploits the symbolic structure of the FOQP, in essence, by appealing to the strengths of the ADD representation. The reader should note that much of this discussion is predicated on problems having considerable logical structure, as is often the case in real-world problems involving relations and properties.

Recall from the previous discussion that in the presence of cache, (compact) ADDs translate to very fast matrix-vector multiplications. Moreover, from Corollary 4, vector operations are efficiently implementable, which implies that given an ADD for A , x and b , the computation of the residual $y = Ax - b$ is more efficient than its matrix counterpart [11]. Analogously, a descent along a direction $(x_k = x_{k-1} + \alpha \Delta x)$ given ADDs for x_{k-1} and Δx has the same run-time complexity as when performed on dense vectors.

We remark that for ADD-based procedures to be efficient, we need to respect the variable ordering implicit in the ADD. Therefore, the solver strategy rests on the following constraints: i) *the engine must manipulate the matrix only through recursive-descent arithmetical operations, such as matrix-vector multiplications (matvecs, for short)*; and ii) *operations must manipulate either the entire matrix or those submatrices corresponding to cofactors (i.e. arbitrary submatrices are non-trivial to access)*.

We will now investigate a method that satisfies these requirements. We proceed in two steps. In step 1, we will demonstrate that solving a QP can be reduced to solving a sequence of linear equations over A . Next, in step 2, we will make use of an iterative solver that computes numerical solutions by a sequence of residuals and vector algebraic operations. As a result, we will obtain a method that fully utilizes the strengths of ADDs. Due to space constraints, we will not be able to discuss the construction in full detail, and so we sketch the main ideas that convey how ADDs are exploited.

Step 1: From linear programs to linear equations. A prominent solver for QPs in standard form is the primal-dual barrier method, see *e.g.* [27], sketched in Alg. 2a. This method solves a perturbed version of the first-order necessary conditions (KKT conditions) for QP:

$$Ax = b, \quad -Qx + A^T y + s = c, \quad XSe = \mu e, \quad (x, s) \geq 0,$$

where $X = \text{diag}(x_1, \dots, x_n)$, $S = \text{diag}(s_1, \dots, s_n)$, and $\mu \geq 0$. The underlying idea is as follows [27, 13]: by applying a perturbed Newton method to the equalities in the above system, the algorithm progresses the current solution along a direction obtained by solving the following linear system:

$$\begin{bmatrix} A & 0 & 0 \\ -Q & A^T & I \\ S & 0 & X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \end{bmatrix} = \begin{bmatrix} b - Ax \\ c - A^T y - s \\ \mu e - Xs \end{bmatrix}, \quad (1)$$

where e is a vector of ones. Observe that besides A and Q , which we already have decision diagrams for, the only new information that needs to be computed is in the form of residuals in the right-hand side of the equation. By performing two pivots, Δx and Δs can be eliminated from the system, reducing it to the so-called *normal equation*:

$$A(Q + \Theta^{-1})^{-1} A^T \Delta y = f, \quad (2)$$

where Θ is the diagonal matrix $\Theta_{ii} = \frac{x_i}{s_i}$ [27, 13]. Once Δy is determined, Δx and Δs are recovered from Δy as $\Delta x = (Q + \Theta)^{-1}(A^T \Delta y - g)$ and $\Delta s = h - A^T \Delta y$, where h and g are obtained from the residuals via vector arithmetic. The reader will note that constructing the left-hand side involves the matrix inverse $(Q + \Theta)^{-1}$. The efficiency of computing this inverse cannot be guaranteed with ADDs, unfortunately; therefore, we assume that either the problem is separable (Q is diagonal), in which case computing this inverse reduces to computing the reciprocals of the diagonal elements (an efficient operation with ADDs), or that we have only box constraints, meaning that A is a diagonal matrix, in which case solving the equation reduces to solving $(Q + \Theta)\Delta y' = A^{-1}f$ and re-scaling. (The general case of going beyond these assumptions is omitted here for space reasons [13].)

The benefit of reformulating (1) into (2) can be appreciated from the observation that 2 becomes positive-semidefinite, which is crucial for solving this system by residuals. To reiterate: the primal-dual barrier method solves a quadratic program iteratively by solving one linear system (2) in each iteration. Constructing the right-hand side of this linear system only requires the calculation of residuals and vector arithmetic, which can be done efficiently with ADDs. Moreover, this system does not require taking arbitrary submatrices of A or Q . Hence, the primal-dual barrier method meets our requirements. Now, let us investigate how 2 can be solved via a sequence of residuals.

Step 2: From linear equations to residuals. To solve (2), we employ the conjugate gradient method [28], sketched in Fig. 2b. Here, the algorithm uses three algebraic operations: 1) matrix-vector products; 2) norm computation ($r^T r$) and 3) scalar updates. From Corollary 4, all of these operations can be implemented efficiently in ADDs. There is, however, one challenge that remains to be addressed. As the barrier method approaches the solution of the QP, the iterates s^k and x^k approach complementary slackness ($x_i s_i = 0$). This means that the diagonal entries of the matrix Θ in (2) tend to either 0 or $+\infty$, making the condition number of (2) unbounded. This is a severe problem for any iterative solver, as the number of iterations required to reach a specified tolerance becomes unbounded. To remedy this situation, Gondzio [13] proposes the following approach: first, the system can be regularized to achieve a condition number bounded by the largest singular value of A . Second, due to the IPM’s remarkable robustness to inexact search directions, it is not necessary to solve the system completely. In practice, decreasing the residual by a factor of 0.01 to 0.0001 has been found sufficient. Finally, a partial pivoted Cholesky factorization can be used to speed-up the convergence. That is, perform a small number k (say 50) Cholesky pivots, and use the resulting trapezoidal matrix as a preconditioner. More details on this can be found in [13]. As demonstrated in [13] this approach does lead to a practical algorithm. Unfortunately, in our setting, constructing this preconditioner requires that we query k rows of N and perform pivots with them. This forces us to partially back down on our requirement (ii), since we need random access to k rows. However, by keeping k small, we can guarantee the ADD in this unfavorable regime will be kept to a minimum.

Thus, we have a method for solving quadratic programs, implemented completely with ADD operations, and much of this work takes full advantage of the ADD representation (thereby, inheriting its superiority). Intuitively, one can expect significant speed-ups over matrix-based methods when the ADDs are compact, *e.g.* arising from structured (in a logical sense) problems.

6 Empirical Illustration

Here, we aim to investigate the empirical performance of our ADD-based interior point solver. There are three main questions we wish to investigate, namely: **(Q1)** in the presence of symbolic structure, does our ADD-based solver perform better than its matrix-based counterpart? **(Q2)** On structured sparse problems, does solving with ADDs have advantages over solving with sparse matrices? And, **(Q3)**, can the ADD-based method handle dense problems as easily as sparse problems?

To evaluate the performance of the approach, we implemented the entire pipeline described here, that is, a symbolic environment to specify QPs, a compiler to ADDs, based on the popular CUDD package, and the symbolic interior-point method described in the previous section.

To address Q1 and Q2, we applied the symbolic IPM on the problem of computing the value function of a family of Markov decision processes used in [21]. These MDPs concern a factory agent whose task is to paint two objects and connect them. A number of operations (actions) need to be performed on these objects before painting, each of which requires the use special of tools, which are possibly available. Painting and connecting can be done in different ways, yielding results of various quality, and each requiring different tools. The final product is rewarded according whether the required level

of quality is achieved. Since these MDPs admit compact symbolic representations, we consider them good candidates to illustrate the potential advantages of symbolic optimization. The computation of an MDP value function corresponds to the following LP: $\min. \sum_{s:\text{state}(s)} v(s)$, s.t. $\{s : \text{state}(s), a : \text{act}(a)\} : v(s) \geq \text{rew}(s) + \gamma \sum_{s':\text{state}(s')} \text{tprob}(s, a, s') v(s')$, where s, s', a are vectors of Boolean variables, state is a Boolean formula whose models are the possible states of the MDP, act is a formula that models the possible actions, and rew and tprob are pseudo-Boolean functions that model the reward and the transition probability from s to s' under the action a . We compared our approach to a matrix implementation of the primal-dual barrier method, both algorithms terminate at the same relative residual, 10^{-5} . The results are summarized in the following table.

Problem Statistics				Symbolic IPM		Ground IPM
name	#vars	#constr	$nnz(A)$	ADD	time[s]	time[s]
factory	131.072	688.128	4.000.000	1819	6899	516
factory0	524.288	2.752.510	15.510.000	1895	6544	7920
factory1	2.097.150	11.000.000	59.549.700	2406	34749	159730
factory2	4.194.300	22.020.100	119.099.000	2504	36248	$\geq 48\text{hrs.}$

The symbolic IPM outperforms the matrix-based IPM on the larger instances. However, the most striking observation is that the running time depends mostly on the size of the ADD, which essentially translates to scaling sublinear in the number of nonzeros in the constraints matrix. To the best of our knowledge, no generic method, sparse or dense, can achieve this scaling behavior. This provides an affirmative answer to Q1 and Q2.

Our second illustration concerns the problem of compressed sensing (CS). That is, we are interested in recovering the sparsest solution to the problem $\|Ax - b\|^2$, where $A \in \mathbb{R}^{m \times n}$ and $n \gg m$ where n is the signal length and m the number of measurements. While this is a hard combinatorial problem, if the matrix A admits the so-called Restricted Isometry Property, the following convex problem (Basis Pursuit Denoising - BPDN): $\min_{x \in \mathbb{R}^n} \tau \|x\|_1 + \|Ax - b\|_2^2$ recovers the exact solution. The matrices typically used in CS tend to be dense, yet highly structured, often admitting an $O(n \log n)$ Fast Fourier Transform-style recur-and-cache matrix-vector multiplication scheme, making BPDN solvers efficient. A remarkable insight is that these fast recursive transforms are very similar to the ADD matrix-vector product. In fact, if we take A to be the Walsh matrix $W_{\log(n)}$ which admits an ADD of size $O(\log n)$, the ADD matrix-vector product resembles the Walsh-Hadamard transform. Moreover, this compact ADD is extracted automatically from the symbolic specification of the Walsh matrix. We illustrate this in the following experiment. We apply the Symbolic IPM to the BPDN reformulation of [29], with the Walsh matrix specified symbolically, recovering random sparse vectors. We compare to the reference MATLAB code provided by Gondzio et al, which we have modified to use MATLAB's implementation on the fast Walsh-Hadamard transform. The task is to recover a sparse random vector with $k = 50$ normally distributed nonzero entries. Results below.

Problem Statistics		Symbolic IPM	FWHT IPM [29]
n	m	time[s]	time[s]
2^{12}	2^{10}	8.3	18
2^{13}	2^{11}	20.2	28.2
2^{14}	2^{12}	46.5	43.9
2^{15}	2^{13}	99.2	65.7

While the method does not scale as well as the hand-tailored solution, we demonstrate that the symbolic approach can handle dense matrices reasonably well, supporting the positive answer of Q3.

7 Conclusions

A long-standing goal in machine learning and AI, which is also reflected in the philosophy on the *democratization of data*, is to make the specification and solving of real-world problems simple and natural, possibly even for non-experts. To this aim, we considered first-order logical mathematical programs that support individuals, relations and connectives, and developed a new line of research of symbolically solving these programs in a generic way. In our case, a matrix-free interior point method was argued for. Our empirical results demonstrate the flexibility of this research direction.

The most interesting avenue for future work is to explore richer modeling languages paired with more powerful circuit representations.

Acknowledgements The authors would like to thank the anonymous reviewers for their feedback. The work was partly supported by the DFG Collaborative Research Center SFB 876, project A6.

References

- [1] K. P. Bennett and E. Parrado-Hernández, “The interplay of optimization and machine learning research,” *JMLR*, vol. 7, pp. 1265–1281, 2006.
- [2] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1994.
- [3] K. Murphy, *Machine learning: a probabilistic perspective*. The MIT Press, 2012.
- [4] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov, “BLOG: Probabilistic models with unknown objects,” in *Proc. IJCAI*, 2005, pp. 1352–1359.
- [5] M. Richardson and P. Domingos, “Markov logic networks,” *Machine learning*, 2006.
- [6] D. McAllester, B. Milch, and N. D. Goodman, “Random-world semantics and syntactic independence for expressive languages,” MIT, Tech. Rep. MIT-CSAIL-TR-2008-025, 2008.
- [7] M. Grant and S. Boyd, “Graph implementations for nonsmooth convex programs,” in *Recent Advances in Learning and Control*, ser. Lecture Notes in Control and Information Sciences, 2008, pp. 95–110.
- [8] P. M. Domingos, “A few useful things to know about machine learning,” *Commun. ACM*, vol. 55, no. 10, pp. 78–87, 2012.
- [9] A. Friesen and P. Domingos, “Recursive decomposition for nonconvex optimization,” in *IJCAI*, 2015.
- [10] M. Chavira and A. Darwiche, “On probabilistic inference by weighted model counting,” *Artif. Intell.*, vol. 172, no. 6-7, pp. 772–799, 2008.
- [11] M. Fujita, P. McGeer, and J.-Y. Yang, “Multi-terminal binary decision diagrams: An efficient data structure for matrix representation,” *Formal Methods in System Design*, vol. 10, no. 2, pp. 149–169, 1997.
- [12] E. M. Clarke, M. Fujita, and X. Zhao, *Representations of Discrete Functions*. Boston, MA: Springer US, 1996, ch. Multi-Terminal Binary Decision Diagrams and Hybrid Decision Diagrams, pp. 93–108.
- [13] J. Gondzio, “Matrix-free interior point method,” *Comp. Opt. and Appl.*, vol. 51, no. 2, pp. 457–480, 2012.
- [14] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Mathematical Programming Language*. The Scientific Press, San Francisco, CA, 1993.
- [15] S. Wallace and W. Ziemba, Eds., *Applications of Stochastic Programming*. SIAM, Philadelphia, 2005.
- [16] P. Domingos, S. Kok, H. Poon, M. Richardson, and P. Singla, “Unifying logical and statistical AI,” in *Proc. AAAI*, 2006, pp. 2–7.
- [17] W. Yih and D. Roth, “Global inference for entity and relation identification via a linear programming formulation,” in *An Introduction to Statistical Relational Learning*. MIT Press, 2007.
- [18] D. Klabjan, R. Fourer, and J. Ma, “Algebraic modeling in a deductive database language,” in *11th INFORMS Computing Society Conference*, 2009.
- [19] K. Kersting, M. Mladenov, and P. Tokmakov, “Relational linear programming,” *Artificial Intelligence Journal (AIJ)*, vol. OnlineFirst, 2015.
- [20] G. Gordon, S. Hong, and M. Dudík, “First-order mixed integer linear programming,” in *UAI*, 2009, pp. 213–222.
- [21] J. Hoey, R. St-Aubin, A. Hu, and C. Boutilier, “Spudd: Stochastic planning using decision diagrams,” in *UAI*, 1999, pp. 279–288.
- [22] Z. Zamani, S. Sanner, K. V. Delgado, and L. N. de Barros, “Robust optimization for hybrid mdps with state-dependent noise,” in *IJCAI*, 2013.
- [23] H. Cui and R. Khadon, “Online symbolic gradient-based optimization for factored action mdps,” in *IJCAI*, 2016.
- [24] H. Enderton, *A mathematical introduction to logic*. Academic press New York, 1972.
- [25] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.
- [26] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: a modeling language for mathematical programming*. South San Francisco: Scientific Press, 1993.

- [27] F. Potra and S. J. Wright, “Interior-point methods,” *Journal of Computational and Applied Mathematics*, vol. 124, pp. 281–302, 2000.
- [28] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.
- [29] K. Fountoulakis, J. Gondzio, and P. Zhlobich, “Matrix-free interior point method for compressed sensing problems,” *Mathematical Programming Computation*, vol. 6, no. 1, pp. 1–31, 2014.